Using Abstraction with Interaction Sequences for Interactive System Modelling

Jessica Turner, Judy Bowen, and Steve Reeves

University of Waikato, Hamilton, New Zealand
jdt11@students.waikato.ac.nz
{jbowen,stever}@waikato.ac.nz

Abstract. Interaction sequences can be used as an abstraction of an interactive system. We can use such models to consider or verify properties of a system for testing purposes. However, interaction sequences have the potential to become unfeasibly long, leading to models which are intractable. We propose a method of reducing the state space of such sequences using the *self-containment* property. This allows us to hide (and subsequently expand) some of the model describing parts of the system not currently under consideration. Interaction sequences and their models can therefore be used to control the state space size of the models we create as an abstraction of an interactive system.

Keywords: Interaction Sequences · Interactive System Testing · Formal Methods.

1 Introduction

As part of a sound software engineering development process, interactive systems should be tested thoroughly to ensure behaviour is as expected. In the process of developing and maintaining safety-critical interactive systems (systems in which failure can lead to serious injury or even fatalities [15, 12]) this is particularly important. Models and model-based testing are useful techniques to employ when testing interactive systems as they focus on different aspects of the system, the functionality or the usability, which provides flexibility when designing tests.

In order to model the system behaviour, interaction sequences can be used as a simple abstraction. An interaction sequence is the series of steps a user can take to perform a certain task or arbitrarily explore an interactive system. We can derive these sequences at different points in the development life-cycle, for example from formal specifications, system prototypes or from implementations. Interaction sequences can take many different forms depending on the specific technique being used and the required level of abstraction. In our work we describe the sequences in terms of system states, widgets of the user interface (UI), user tasks, or combinations of these. We formalise these sequences using Presentation Models (PM) (see [4]) and Finite State Automata (FSA).

Regardless of how the sequences are formalised, conceptually we can think of them as never-ending and they can also be combined in an inexhaustible number

of ways. This is reflected in the models of sequences as an increased number of states which can lead to intractably large models — the state explosion problem. The main contribution of this paper is an approach using abstraction of parts of a sequence to address this problem. We define the property of *self-containment* and use this to abstract parts of the model into an *abstract state*, consequently reducing the state space. By abstracting sequences using this property we are able to hide certain parts of the model, however we can also retrieve this information if required by expanding the abstract state without loss of information. Therefore, we may be able to reduce and expand the state space using the self-containment property, providing the ability to constrain the size of the model.

2 Background and Related Work

In this research, our focus is on modelling interaction sequences, specifically task-widget based sequences (we will discuss different types of sequences in more depth later). Several approaches to modelling interaction sequences in the domain of interactive system testing exist. A common theme between different approaches is how to constrain or limit the models so that they remain tractable. We discuss the most relevant techniques to our work here.

The use of directed graphs is a popular visualisation for many of the techniques we will discuss here, such as Event Flow Graphs (EFG) [10], FSA (used here interchangeably with Finite State Machines (FSM)) [18, 8, 19, 13, 7, 16, 1], and hierarchical Task Models [11, 2, 5]. Directed graphs establish specific paths through a graph which allow us to traverse specific orderings. They allow us to view and easily understand how we can generate sequences of varying lengths.

There are different ways in which state explosion in directed graphs can be managed. One approach is to limit by sequence length, which is utilised by Nguyen *et al.* in the creation of their testing tool GUITAR [10]. They utilise interaction sequences to describe systems using EFGs. All sequences of a given length (such as two) are then generated and they systematically explore these sequences. Constraining sequences to a defined length gives control over the state space size, however, it does also potentially hide behaviours that could be exposed by longer sequences, or combinations of longer sequences.

Finite state automata, or more specifically Mealy machines, are used to model systems for testing purposes by making certain assumptions about the System Under Test (SUT), and modelling the system based on input/output pairs [18]. To address the state explosion problem an extended finite state machine (EFSM) is used which has variables to store important information. For example, a timeout counter variable can be used instead of three duplicated timeout states. This reduces the number of states required to model the SUT and restricts the length of the sequences. It is possible to have lengthy sequences with no duplication and using an EFSM does not guarantee constraining models to a tractable size.

Interaction sequences are also used in some testing approaches where well-known traversal algorithms, or variations of these, are used to explore their models. This is another approach which focuses on restricting the sequence length

to those generated by specific traversal algorithms. For example, Salem presents an approach where an FSA is converted so they can be explored using the UPPAAL model checker, this method allows them to avoid direct state-explosion [13]. In [7] Huang *et al.* use weight based methods to calculate paths of a specified length to traverse through the models. Essentially these approaches, and others like them, allow the traversal algorithm to "trim" the model. For example, a weighted strategy only traverses sequences which are more likely to occur based on probability metrics. This type of strategy only works under certain conditions for specific types of software (such as GUI-based applications as in [7]) and further abstraction is often required to reduce the model's complexity.

The symmetry property is introduced in [8] by Ip and Dill, which can be applied to directed graphs to simplify them. They argue if a series of states results in the same output, it does not matter which path is taken, as the result is the same. This use of symmetry could, "help to reduce even infinitely long graphs", and as a consequence reduce the overall sequence length. However, we found that symmetry is not common in interaction sequences. Complete Interaction Sequences (CIS) are a way to model the responsibilities (what the system should allow the user to perform) of an interface rather than the user actions [19]. Using FSA to model these responsibilities still results in the state explosion problem. In order to reduce the number of states, strongly connected components, or symmetric components, are identified and abstracted into a 'super' state. This gives a significant reduction in the number of sequences, as well as their length. While these interaction sequences differ from those we present (they consider sequences at a higher level of abstraction) the identification of specific components as the basis for abstraction is relevant to our work and has informed our approach.

Another way of constraining interaction sequences is to focus on specific tasks. This allows us to consider only sequences aimed at satisfying specific goals (although many different sequences may satisfy the same task). Since the sequences used are based on tasks and widgets, the extensive literature on task modelling is relevant. Particularly those based around tools for modelling interactive systems such as CTT [11] and HAMSTERS [2]. These task models focus on the set of steps a user takes to complete a certain task, and in this respect form the basis for own approach. The main point of difference from the modelling perspective is that while task models typically view the system relationships at a higher level and hierarchically decompose tasks into smaller and smaller steps, we use the task as a mechanism for composing user actions into specific groupings, which enables us to limit interaction sequence length (combining the two methods of length and tasks). We link these task definitions to system actions specifically via the widget descriptions.

3 Interaction Sequences

We have identified three perspectives which can be used as the basis for interaction sequences, these are state-based; task-based; and widget-based. We can use these individually, or in combination with each other to build sequences.

State-based sequences are created by identifying the different states (which may relate to composite states, windows, dialogs or modes) available in a system and how the user is able to transition between these states. Task-based sequences are created by taking a goal (as a task description) the user wishes to achieve, and then listing the interactions (or set of interactions) it takes to achieve that goal. Lastly, widget-based sequences are created by identifying the different widgets that are available in the system and the actions associated with those widgets.

Our larger goal for modelling interaction sequences is to adapt them for interactive system testing purposes. This leads to the following requirements for our sequences (we discuss each of these requirements next):

1. We must be able to automatically generate sequences of varying lengths so that the testing process is faster.
2. We must be able to constrain the sequence length in order to avoid the state explosion problem.
3. The sequences must allow us to clearly identify why the system did not behave as expected, for example by producing counter-examples.

3.1 Automatic generation

We can already automatically generate interaction sequences of varying lengths using the Presentation Models (PM) of the SUT. PMs provide us with an abstract view of the interactive component of an interactive system with widgets described as triples of the form: "$((WidgetName, WidgetCategory, (Behaviour(s)$)". To build sequences we begin by modelling the PMs of the SUT, taking into account the widgets and their related actions, for example "Button1" has the action "Press". In order to be able to build these models and their respective sequences, we must have a thorough understanding of the system. It is expected in a good engineering design process this knowledge is readily available from task models, user-centred design artefacts, specifications etc. We make assumptions about the sequence based on internal values of the system (for example, we may want to generate a sequence where a counter variable is 10) and generate steps of the form: "$< action >< widget >< number\ of\ interactions >$". Once we have a generated sequence we can then model this as an FSA.

We use FSA to model the sequences due to their simplicity and the advantage of being able to draw on existing, well-defined, theory (other approaches have used FSA to model interaction systems see [16, 1, 5] for similar advantages). This enables us to manipulate FSA using standard techniques, such as removal of non-determinism or minimisation. We can easily combine multiple FSA by making use of task ordering knowledge and techniques, such as union or concatenation.

We can generate subsequences of any given sequence by traversing its FSA via different traversal algorithms. When sequences are adapted for testing this is a useful characteristic of the sequence models, as it allows us to explore variations of particular tasks and exploring such variations is more likely to expose errors. This also mimics users' behaviour, in that they typically do not always follow a pre-defined sequence for a particular task if there are several alternatives.

It is typical in interaction sequences to focus mainly on either direct (see [3, 17]) or response (see [9, 14]) actions. Direct actions are the literal actions performed by the user, for example "Press Ok 1". Response actions are the actions that the user will perform in response to a change in the system, for example "Observe Display". In this work we use both direct and response actions to create a complete set of actions for our sequences.

3.2 Constraining Sequence Length

The focus of this paper is to address this second requirement, that is to lessen the state explosion problem by constraining sequence length. When we first began using interaction sequences for larger and more complex interactive systems we found that using existing theory, such as removal of non-determinism and minimisation, was not enough alone to ensure tractability. Using these techniques resulted in a loss of information in the models, and thus the meaning of the behaviour of the sequence changed. Therefore, we needed a technique which would allow us to hide information, or rather abstract it.

Our first attempt to solve this was to focus solely on task-widget based sequences. Widgets allow us to divide the sequence into steps based on the interactions with those widgets, this allow us to describe sequences consistently. The simplest way to constrain a sequence which "never ends" is to limit the length of that sequence, tasks allow us to do this as every task has a defined "end point" or "goal". From experimentation with different types of models and sequences we found this did not provide a solution. The reason for this being that it resulted in a loss of information about the interaction sequence and its behaviour. The use of FSA to model task-widget based sequences reduces the sequence length further, as it constrains us to subsets of sequences for specific tasks, but it is still not enough to fully solve the state explosion problem. The contribution of this paper is, therefore, a method to address this.

3.3 Using Interaction Sequences for Testing

This requirement further influenced the choice of sequences to task-widget based. The task-based sequences on their own were too "restrictive" in the sense that they did not allow for easy generation while the widget-based sequences were too "free" (allowing for never ending sequences), hence the need for the combination. The state-based sequences have the potential to unintentionally hide widgets of the system which do not have an observable effect on state, resulting in poor coverage of the system behaviour, and for this reason would not be appropriate to use either alone or in combination with the other types. Requirement three will be addressed in future work and we do not discuss this further beyond the implications it has for the work we describe.

4 Definitions

In this research FSA are used to model interaction sequences. Our purpose is to make these models more tractable and therefore we introduce 'the self-containment property'. In what follows we define: the machines as a variation of traditional FSA (def. 1); the self-containment property (def. 4); abstraction (def. 7); and expansion of these machines (def. 8) also supporting definitions for: paths (def. 2); connectedness (def. 3); alphabet function (def. 5); override function (def. 6). We follow this in the next section with lemmas (and their proofs) to show that these definitions have the useful properties we expect and that they have captured the properties necessary to address the state explosion problem.

Definition 1. *A* finite state automaton (FSA) *is of the form* $M \stackrel{def}{=} (Q, \Sigma, \delta, S, F)$ *where:*

1. Q *is a finite set of* states,
2. Σ *is a finite set of symbols, the* alphabet accepted *by* M,
3. δ *is a finite set of triples which defines the* transitions *of machine* M, *i.e. given states* $q, q' \in Q$, *input* $x \in \Sigma$, *we can denote each transition as* (q, x, q'),
4. S *is the set of* start states *and* $S \subseteq Q$,
5. F *is the set of* final (accepting) states *and* $F \subseteq Q$.

Definition 2. *Given a finite state automaton* $M = (Q, \Sigma, \delta, S, F)$, *a* path ρ *from* $q \in Q$ *to* $q' \in Q$ *is a sequence of transitions from* δ *such that* ρ *is the empty sequence* $< >$, *or* ρ *has first element* $(q, x, q'') \in \delta$ *and the remainder of* ρ *is a path from* q'' *to* q'.

If a path exists between two states $q, q' \in Q$ *we say that* q' *is* reachable *from* q.

Definition 3. *A FSA is* connected *iff every state is reachable from a start state.*

Definition 4. *Given machine* $M = (Q, \Sigma, \delta, S, F)$ *we define a machine* $M_s \stackrel{def}{=} (Q_s, \Sigma_s, \delta_s, S_s, F_s)$ *which is* self-contained with respect to M *iff:*

1. $Q_s \subseteq Q$, $\Sigma_s \subseteq \Sigma$, $\delta_s \subseteq \delta$,
2. M_s *is closed with respect to* M, *which means that if any transition in* δ *starts and ends in* Q_s *then it is in* δ_s *too:* $\delta_s = \{(q_s, x, q'_s) | (q_s, x, q'_s) \in \delta \wedge q_s, q'_s \in Q_s\}$,
3. *The only transitions of* M *that start outside* M_s *and end inside* M_s *are those that end in start states of* M_s*: for all* $(q, x, q') \in \delta$, *if* $q \in Q \setminus Q_s$ *and* $q' \in Q_s$ *then* $q' \in S_s$,
4. *The only transitions of* M *that start inside* M_s *and end outside* M_s *are those that start in final states of* M_s*: for all* $(q, x, q') \in \delta$, *if* $q \in Q_s$ *and* $q' \in Q \setminus Q_s$ *then* $q \in F_s$.

Definition 5. *There is an* alphabet function *such that, for any machine* $M = (Q, \Sigma, \delta, S, F)$ *we have* $\alpha(\delta) \stackrel{def}{=} \{x | (q, x, q') \in \delta\}$.

Definition 6. *For any machine $M = (Q, \Sigma, \delta, S, F)$ we can override its set of transitions δ as follows with the* override function*:*

$$\overset{P}{p'}\delta^{Q}_{q'} \overset{def}{=} \left\{ \begin{array}{l} (p'x, r'),\ if\ r \in P \\ (r, x, r'),\ otherwise \end{array} \middle|\ (r, x, r') \in \delta' \right\}$$

where

$$\delta' \overset{def}{=} \left\{ \begin{array}{l} (r, x, q'),\ if\ r' \in Q \\ (r, x, r'),\ otherwise \end{array} \middle|\ (r, x, r') \in \delta \right\}$$

Note: In what follows, we are dealing specifically with interaction sequences, thus a FSA will always be connected, however, the proofs do not rely on this. We also assume that a FSA's alphabet is exactly the set of symbols that label its transitions, i.e. for all FSAs $(Q, \Sigma, \delta, S, F)$ we have $\alpha(\delta) \overset{def}{=} \Sigma$. **End note.**

Definition 7. *Given machine $M = (Q, \Sigma, \delta, S, F)$ where $S \neq \emptyset$ and $F \neq \emptyset$ (we call M the* machine abstracted on*), machine $M_s = (Q_s, \Sigma_s, \delta_s, S_s, F_s)$ where M_s is self-contained with respect to M, and an* abstract state *x where $x \notin Q, Q_s$ then an* abstract machine *$M_a \overset{def}{=} (Q_a, \Sigma_a, \delta_a, S_a, F_a)$ where:*

1. $Q_a = (Q \backslash Q_s) \cup \{x\}$,
2. $\Sigma_a \subseteq \Sigma$,
3. $\delta_a = \overset{F}{x}(\delta \setminus \delta_s)^{S}_{x}$,
4. $(S \cap Q_s = \emptyset \implies S_a = S) \wedge (S \cap Q_s \neq \emptyset \implies S_a = \{x\})$,
5. $(F \cap Q_s = \emptyset \implies F_a = F) \wedge (F \cap Q_s \neq \emptyset \implies F_a = \{x\})$.

The abstract machine is essentially the original machine we started with except with the removal of the self-contained machine. However, this would result in a machine which is not connected, indicating a non-connected interaction sequence. This would be a confusing model of a sequence as it would be unclear how to process a path through the states which were originally connected to the self-contained machine. Therefore, we introduce the abstract state to indicate that an abstraction has taken place and at which point this occurred. The transitions that originally finished and started in the the self-contained machine start and final states are then overridden to reflect this change.

Definition 8. *Given abstract machine $M_a = (Q_a, \Sigma_a, \delta_a, S_a, F_a)$ with abstract state $x \in Q_a$ and any machine $M = (Q, \Sigma, \delta, S, F)$ with $x \notin Q$, there is a machine M_b, which we call the* expansion of M_a with respect to M*, and $M_b \overset{def}{=} (Q_b, \Sigma_b, \delta_b, S_b, F_b)$ where:*

1. $\Sigma_b = \Sigma_a \cup \Sigma$,
2. $Q_b = (Q_a \backslash \{x\}) \cup Q$,
3. $\delta_b = \delta \bigcup_{s \in S, f \in F} (\overset{\{x\}}{f}(\delta_a)^{\{x\}}_{s})$, *which is to say x as a "from" state in a transition is replaced by the final states of M, and x as the "to" state in any transition is replaced by the start states of M,*
4. *If S_a contains only x then S_b contains only s. Otherwise $S_b = S_a$,*
5. *If F_a contains only x then F_b contains only f. Otherwise $F_b = F_a$.*

At some point we may wish to explore the sequence in the self-contained machine, therefore we needed a way to expand the abstract state. Definition 8 shows how we can correctly expand this state, allowing us to reconstruct our original machine. As a result we can reduce and expand the state space.

5 Results

In this section we will prove some results that give some evidence that our definitions correctly capture our intuitions.

Lemma 1. *For any machine* $M = (Q, \Sigma, \delta, S, F)$ *with* $s, f \notin Q$*, there is an equivalent machine* $M_c \stackrel{def}{=} (Q_c, \Sigma_c, \delta_c, S_c, F_c)$ *where:*

1. S *is not a singleton set and*
   (a) $Q_c = Q \cup \{s\}$,
   (b) $\Sigma_c = \Sigma \cup \{\epsilon\}$ *where* ϵ *is the blank symbol,*
   (c) $\delta_c = \delta$ *and for all* $(q, x, q') \in \delta_c$*, if* $q \in S$ *then* $\delta_c = \delta_c \cup (s, \epsilon, q)$,
   (d) $S_c = \{s\}$,
   (e) $F_c = F$.
2. F *is not a singleton set and*
   (a) $Q_c = Q \cup \{f\}$,
   (b) $\Sigma_c = \Sigma \cup \{\epsilon\}$,
   (c) $\delta_c = \delta$ *and for all* $(q, x, q') \in \delta_c$*, if* $q' \in F$ *then* $\delta_c = \delta_c \cup (q', \epsilon, f)$,
   (d) $S_c = S$,
   (e) $F_c = \{f\}$.

Proof: Section 2.2 [6, p. 26] states that a string w with ϵs (ϵ representing the blank symbol) in is equivalent to w. Therefore, by theorem 3.8 from [6, p. 65] the new machine is equivalent to M as it accepts the same language.
□

Task-widget based interaction sequences have a defined single start and end point to the sequence due to the nature of tasks, and thus have singleton start and final state sets. However, we could have machines which do not. Lemma 1 shows that for any machine there is an equivalent machine with singleton start and final state sets, thus we do not have to include this as a restriction.

Lemma 2. *Given a machine* $M = (Q, \Sigma, \delta, S, F)$*,* M *is self-contained with respect to itself.*

Proof:

1. Immediate.
2. Immediate.
3. There are no states of M outside M, therefore implication is true (since false implies anything, *ex falso quod libet*).
4. Similarly to 3.

□

Lemma 2 proves that for any given machine it is self-contained with respect to itself. This addresses the state explosion problem in the most extreme case as we can now take any machine and reduce the state space to exactly one state, the abstract state. However, this also results in loss of all information for that machine as it is hidden inside this abstract state. While this solves the state explosion problem, it is not particularly useful or interesting, especially not in consideration of adapting the sequences and their consequent models for testing.

Our main result is that, under certain circumstances, we can take a machine M, abstract it with respect to machine M_s (where M_s is self-contained with respect to M) to get abstract machine M_a, and then expand M_a with respect to M_s to get machine M again. While we have all of the component parts in the definitions above, there is still a crucial relationship amongst the various machines that we are missing, and this is that we have, of course, to be able to re-connect the start and final states as originally intended when expanding the abstract machine. The definitions so far, while allowing re-connection, lose crucial information about start and final states. The property that we require for our main result ensures that this information can be recovered. The property is that if *any* state of the self-contained machine M_s is also a start state of the machine M it is self-contained with respect to, then the start states of the self-contained machine *must be* the start states of the original machine. Essentially we need this as we use the start and final states as "markers" to show how the various machines fit together properly when we do the expansion. It turns out that this also requires that all the machines involved have singleton start and final state sets, but we already know (by lemma 1) that this is not a restriction.

All this leads to needing the following:

Definition 9. *Given machine* $M = (Q, \Sigma, \delta, S, F)$ *and machine* $M_s = (Q_s, \Sigma_s, \delta_s, S_s, F_s)$ *which is self-contained with respect to* M*, then* M *and* M_s *have the* SF property *iff: if* any *state of* M_s *is also a start state* M*, then the start states of* M_s must be *the start states of* M*, i.e.*

$$Q_s \cap S \neq \emptyset \implies S_s = S$$

and similarly for final states

$$Q_s \cap F \neq \emptyset \implies F_s = F$$

Note that in our case where we can assume all machines have singleton start and final state sets, these conditions simplify to

$$s \in Q_s \implies s_s = s$$

and

$$f \in Q_s \implies f_s = f$$

because $S = \{s\}, F = \{f\}, S_s = \{s_s\}$ *and* $F_s = \{f_s\}$.

Lemma 3. *Let $M = (Q, \Sigma, \delta, \{s\}, \{f\})$ be any machine for modelling interaction sequences and $M_s = (Q_s, \Sigma_s, \delta_s, \{s_s\}, \{f_s\})$ be a self-contained machine with respect to M. We are assuming without loss of generality that machines M and M_s have singleton start and final sets, by lemma 1. We require that M and M_s have the SF property (definition 9). Further, let $M_a = (Q_a, \Sigma_a, \delta_a, S_a, F_a)$ be an abstract machine with abstract state $x \notin Q, Q_s$, where M_s is the machine abstracted on. Finally, we assume a machine $M_b = (Q_b, \Sigma_b, \delta_b, S_b, F_b)$ which is the expansion of M_a with respect to M_s. Then our result is that machine M_b is equivalent to machine M.*

Proof

We have

$$\delta_a = {}^{\{f_s\}}_{x}(\delta \setminus \delta_s)^{\{s_s\}}_{x} \qquad \text{from definition 7} \quad (1)$$

and

$$\begin{aligned}
\delta_b &= \delta_s \cup {}^{\{x\}}_{f_s}(\delta_a)^{\{x\}}_{s_s} && \text{from definition 8} && (2)\\
&= \delta_s \cup {}^{\{x\}}_{f_s}(\ {}^{\{f_s\}}_{x}(\delta \setminus \delta_s)^{\{s_s\}}_{x})^{\{x\}}_{s_s} && \text{substituting from 1} && (3)\\
&= \delta_s \cup (\delta \setminus \delta_s) && \text{over-riding and then reversing} && (4)\\
&= \delta && \delta_s \subseteq \delta \text{ from definition 4 and set theory} && (5)
\end{aligned}$$

So also

$$\begin{aligned}
\Sigma &= \alpha(\delta) && \text{by our Note above} && (6)\\
&= \alpha(\delta_b) && \text{by substitution and (2)-(5)} && (7)\\
&= \Sigma_b && \text{by our Note above} && (8)
\end{aligned}$$

Then

$$\begin{aligned}
Q_b &= (Q_a \setminus \{x\}) \cup Q_s && \text{by definition 8} && (9)\\
&= (((Q \setminus Q_s) \cup \{x\}) \setminus \{x\}) \cup Q_s && \text{by definition 7 } Q_a = (Q \setminus Q_s) \cup \{x\} && (10)\\
&= (Q \setminus Q_s) \cup Q_s && \text{by definition 7 } x \notin Q, Q_s && (11)\\
&= Q && Q_s \subseteq Q \text{ from definition 4 and set theory} && (12)
\end{aligned}$$

Turning to the start states, recall from definition 8 if S_a contains only x then S_b contains only s_s. Otherwise $S_b = S_a$. Within those cases each has to consider whether or not $s \in Q_s$. We proceed by nested cases.

Assume S_a contains only x, so $S_a = \{x\}$. (13a)

Now we have further cases depending on $s \in Q_s$.

Assume $s \in Q_s$ (13ba)

$\{s\} = \{s_s\}$ by definition of 9 and 13ba (13bb)

$= S_b$ by consequence of 13a and definition 8 (13bc)

Assume $s \notin Q_s$ (13ca)

$\{s\} = S_a$ by def. 7, since 13ca means $S \cap Q_s = \emptyset$ (13cb)

$= \{x\}$ by 13a (13cc)

contradiction definition 7 requires $x \notin Q$, but $s \in Q$ (13cd)

Assume $S_a \neq \{x\}$ (13d)

Now we have further cases depending on $s \in Q_s$

Assume $s \in Q_s$ (13ea)

$S_a = \{x\}$ by definition 7 and 13ea (13eb)

contradiction by 13d (13ec)

Assume $s \notin Q_s$ (13fa)

$\{s\} = S_a$ by 13fa and definition 7 (13fb)

$= S_b$ by 13d and definition 8 (13fc)

By cases (twice) we conclude that $S_b = \{s\}$ (13g)

Finally to the final states, recall that definition 8 gives if F_a contains only x then F_b contains only f_s. Otherwise $F_b = F_a$. Within those cases each has to consider whether or not $f \in Q_s$. We proceed by nested cases.

Assume F_a contains only x, so $F_a = \{x\}$. (13h)

Now we have further cases depending on $f \in Q_s$.

Assume $f \in Q_s$ (13ia)

$\{f\} = \{f_s\}$ by definition of 9 and 13ia (13ib)

$= F_b$ by consequence of 13h and definition 8 (13ic)

Assume $f \notin Q_s$ (13ja)

$\{f\} = F_a$ by def. 7, since 13ja means $F \cap Q_s = \emptyset$ (13jb)

$= \{x\}$ by 13h (13jc)

contradiction definition 7 requires $x \notin Q$, but $f \in Q$ (13jd)

Assume $F_a \neq \{x\}$ (13k)

Now we have further cases depending on $f \in Q_s$

Assume $f \in Q_s$ (13la)
$F_a = \{x\}$ by definition 7 and 13la (13lb)
contradiction by 13k (13lc)

Assume $f \notin Q_s$ (13ma)
$\{f\} = F_a$ by 13ma and definition 7 (13mb)
$= F_b$ by 13k and definition 8 (13mc)

By cases (twice) we conclude that $F_b = \{f\}$ (13n)

We have, in 2-5, 6-8, 9-12, 13g and 13n, that $M = M_b$ as required.
□

6 Infusion Pump Example

In this example we illustrate our main result as proven in Lemma 3 specifically for interaction sequences, in this case for a simplified infusion pump, created in reference to the Alaris GP Volumetric Pump (see figure 1). This simplified version has the functionality to set up an infusion based on duration, time and pump type; start, pause or stop an infusion; and view settings and check the battery life. In total it has six widgets which allow the user to perform different actions, these are the Up, Down, YesStart, NoStop, OnOff buttons, and Display.

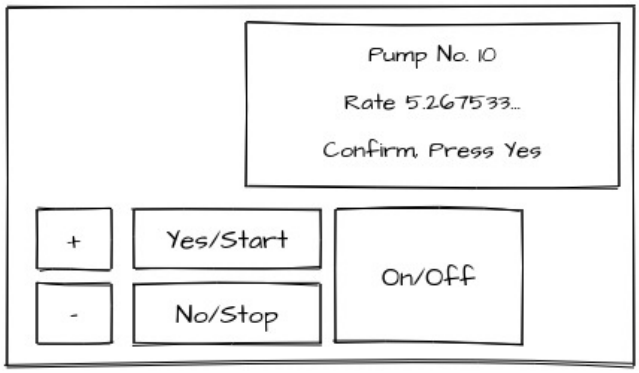

Fig. 1. Wireframe of: Simplified Medical Infusion Pump

We create a task-widget-based sequence for this device. The tasks are: setting up an infusion; starting the infusion; checking the settings; pausing and then

stopping the infusion. Note that a task-based sequence does not need to be based on a single task, as in practice it is common to combine tasks to create more meaningful sequences. To generate the interaction sequence we must make a few assumptions, this is to ensure that the sequence is reproducible and has no ambiguity. In this example we assume that all initial values are set to 0; we begin in the initial state of the system; volume is set to 4ml and duration is set to 2 hours. Using the PM for this example we generate the following sequence.

1. Click YesStart 1.
2. Observe Display 1.
3. Click Up 4.
4. Observe Display 1.
5. Click YesStart 1.
6. Click Up 2.
7. Observe Display 1.
8. Click YesStart 1.
9. Observe Display 1.
10. Click YesStart 1.
11. Observe Display 1.
12. Click YesStart 1.
13. Observe Display 1.
14. Click YesStart 1.
15. Observe Display 1.
16. Click YesStart 1.
17. Observe Display 1.
18. Click NoStop 2.

We can now convert this sequence to an FSA. This involves using definition 1 to construct a well-formed machine $M = \{Q, \Sigma, \delta, S, F\}$. For machine M, Q is the set of widgets used in the sequence and Σ is the set of interactions. δ represents the transitions of the machine in the form (q, x, q') where q is the widget from the previous step, x is the interaction of the current step, and q' is the widget from the current step. If a widget is interacted with more than once, for example "Click Up 4", then this step also has the transition (q', x, q'). The start set S is a singleton set comprising of the state "Initialise" which is a "place holder" to ensure that we have included the initial action performed on the YesStart as a triple in δ. The final set F is a singleton set including the final widget of the final step. Therefore, machine M is as follows:

$Q = \{Initialise, Display, NoStop, YesStart, Up\}$
$\Sigma = \{Click, Observe\}$
$\delta = \{(Initialise, Click, YesStart), (Display, Click, NoStop), (Display, Click,$
$YesStart), (Display, Click, Up), (NoStop, Click, NoStop), (YesStart, Click, Up),$
$(YesStart, Observe, Display), (Up, Click, Up), (Up, Observe, Display)\}$
$S = \{Initialise\}$
$F = \{NoStop\}$

Note that in FSA M we assume that the device is already switched on prior to any interaction. The FSA allows us to generate sequences of varying lengths for a specific task based on the assumptions. This has helped in reducing the number of sequences we explore due to the use of the task to constrain the sequence and consequently the model, in other words the FSA of the sequence.

We now apply the definition of self-containment (def. 4) to this machine to construct machine $M_s = \{Q_s, \Sigma_s, \delta_s, S_s, F_s\}$:

$Q_s = \{Display, YesStart, Up\}$
$\Sigma_s = \{Click, Observe\}$
$\delta_s = \{(Display, Click, YesStart), (Display, Click, Up), (YesStart, Click, Up), (Yes$
$Start, Observe, Display), (Up, Click, Up), (Up, Observe, Display)\}$
$S_s = \{YesStart\}$

$F_s = \{Display\}$

M_s is not the only self-contained machine we can construct using definition 4. As proven in lemma 2 every machine is self-contained with respect to itself and in fact each single state could be a self-contained machine, however as stated previously this would not be particularly useful in terms of the state explosion problem. If we inspect M_s it contains all the widgets associated with setting up and starting the infusion. We are left with a sequence which we assume sets up and begins an infusion correctly, then explicitly pauses and stops that infusion.

To perform the abstraction we create a new FSA $M_a = \{Q_a, \Sigma_a, \delta_a, S_a, F_a\}$ as per definition 7:

$Q_a = \{Initialise, \Omega_0, NoStop\}$
$\Sigma_a = \{Click\}$
$\delta_a = \{(Initialise, Click, \Omega_0), (\Omega_0, Click, NoStop), (NoStop, Click, NoStop)\}$
$S_a = \{Initialise\}$
$F_a = \{NoStop\}$

In this machine we have added an abstract state "Ω_0" representing M_s. In lemma 3 the machine we are abstracting must have singleton start and final states in order to preserve equivalence, in this case M_s satisfies this condition. If required, we could apply lemma 1 to M_s to ensure that this is true.

The abstract sequence for the same task is reduced from 18 steps to two. It is important to remember that this reduction comes from being able to not only contain the other 16 steps in a self-contained machine, but also from specifying a focus for later testing purposes. If we wish to test the setup and start of the infusion we could focus on the self-contained machine M_s, ignoring the last two steps of the original sequence, however the reduction here is significantly smaller.

Using definition 8 we can reconstruct our original machine M by expanding the abstract state. The input transitions to the abstract state are re-directed to the start state of the sub-machine, and the output transitions are now output transitions of the final state of the sub-machine.

The new machine $M_b = \{Q_b, \Sigma_b, \delta_b, S_b, F_b\}$ as per definition 8:

$Q_b = \{Initialise, Display, NoStop, YesStart, Up\}$
$\Sigma_b = \{Click, Observe\}$
$\delta_b = \{(Initialise, Click, YesStart), (Display, Click, NoStop), (Display, Click, Yes$
$Start), (Display, Click, Up), (NoStop, Click, NoStop), (YesStart, Click, Up), (Yes$
$Start, Observe, Display), (Up, Click, Up), (Up, Observe, Display)\}$
$S_b = \{Initialise\}$
$F_b = \{NoStop\}$

As expected from lemma 3 M and M_b are equivalent machines. This result illustrates that even in a small example we can significantly reduce the number of states in a machine of the form in definition 1, thus addressing the state explosion problem. Furthermore, should we wish to revisit the original machine we are able to expand the abstract state, this allows us to hide, rather than

lose, information, which may become important when adapting the sequences for testing purposes. More importantly, this gives control over the size of the state space to reduce and expand as required.

To demonstrate the use of our technique, in this example we show machine M and then build the corresponding abstract machine. However, in practical use we envision that machines will be constructed with abstract states to hide certain parts of an interactive system, which can be modelled later (or not at all). For example, in a safety-critical interactive system we may wish to focus specifically on the safety-critical aspects of that system, we may construct an abstract machine which hides the non-safety-critical aspects in abstract states. We will then be able to use this technique to expand the abstract state if required.

7 Future Work and Conclusions

In this paper we have introduced a new technique for abstracting and expanding states in an FSA representing interaction sequences to provide more control over the state space. We described how we use tasks and widgets to describe interaction sequences and how we formalise them using PMs and FSA. We discussed sequence length and tasks to constrain sequences to avoid intractable models. We also highlighted how this in combination with existing techniques such as FSA minimisation was not enough to address the state explosion problem.

This led to further investigation into abstraction within models to address this problem. The main contribution of this paper was to define the self-containment property and how this is used to further abstract and constrain sequences. Furthermore, we showed how we could expand the abstract state to include the hidden information, allowing us to reduce and expand the state space as required. This not only addressed the state explosion problem but also provided us with greater control over the state space and results in more tractable models.

Our modelling approach is not without limitations, the major concern being we could have a model which contains no self-contained sub-models (beyond the trivial case of abstracting to a single state). In this instance we are not be able to abstract the model further using this method. It is possible that this could occur in a highly inter-connected system and further investigation is required.

Furthermore, while we can use the self-containment property to construct the abstract machine automatically, we cannot know if this abstraction will be useful or not (in terms of adapting the sequences for testing purposes). Keeping in mind that we can abstract an entire machine to a single abstract state, we leave it to human reasoning to determine if abstracting a self-contained machine provides benefits or not from a testing perspective. Future work will involve investigations into adapting this approach for testing and the implications of the abstraction in the testing environment.

References

1. Banu-Demergian, I.T., Stefanescu, G.: Towards a formal representation of interactive systems. Fundamenta Informaticae **131**(3-4), 313–336 (2014)

2. Barboni, E., Ladry, J.F., Navarre, D., Palanque, P., Winckler, M.: Beyond modelling: an integrated environment supporting co-execution of tasks and systems models. In: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems. pp. 165–174. ACM (2010)
3. Bauersfeld, S., Vos, T.E.: Guitest: a java library for fully automated gui robustness testing. In: Proceedings of the 27th ieee/acm international conference on automated software engineering. pp. 330–333. ACM (2012)
4. Bowen, J., Reeves, S.: Formal models for user interface design artefacts. Innovations in Systems and Software Engineering **4**(2), 125–141 (2008)
5. Campos, J.C., Fayollas, C., Gonçalves, M., Martinie, C., Navarre, D., Palanque, P., Pinto, M.: A more intelligent test case generation approach through task models manipulation. Proceedings of the ACM on Human-Computer Interaction **1**(1), 9 (2017)
6. Hopcroft, J.E.: Introduction to Automata Theory, Languages, and Computation. Pearson Education India (1979)
7. Huang, C.Y., Chang, J.R., Chang, Y.H.: Design and Analysis of GUI Test-case prioritization using Weight-based methods. Journal of Systems and Software **83**(4), 646–659 (2010)
8. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal methods in system design **9**(1-2), 41–75 (1996)
9. Martinie, C., Palanque, P., Winckler, M.: Structuring and Composition Mechanisms to Address Scalability Issues in Task Models. In: Human-Computer Interaction–INTERACT 2011, pp. 589–609. Springer (2011)
10. Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.: GUITAR: an Innovative Tool for Automated Testing of GUI-driven Software. Automated Software Engineering **21**(1), 65–105 (2014)
11. Paternò, F., Zini, E.: Applying Information Visualization Techniques to Visual Representations of Task Models. In: Proceedings of the 3rd annual conference on Task models and diagrams. pp. 105–111. ACM (2004)
12. Porrello, A.M.: Death and denial: The failure of the therac-25, a medical linear accelerator. Website (nd), retrieved July 27, 2015 from http://users.csc.calpoly.edu/ jdalbey/SWE/Papers/THERAC25.html
13. Salem, P.: Practical programming, validation and verification with finite-state machines: a library and its industrial application. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 51–60. ACM (2016)
14. Spano, L.D., Fenu, G.: Icett: a responsive visualization for task models. In: Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems. pp. 197–200. ACM (2014)
15. The Economist: When code can kill or cure. Website (June 2012), retrieved December 9, 2015 from http://www.economist.com/node/21556098
16. Thimbleby, H.: Contributing to safety and due diligence in safety-critical interactive systems development by generating and analyzing finite state models. In: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems. pp. 221–230. ACM (2009)
17. Thimbleby, H.: Action graphs and user performance analysis. International Journal of Human-Computer Studies **71**(3), 276–302 (2013)
18. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann-Elsevier Inc. (2010)
19. White, L., Almezen, H., Alzeidi, N.: User-based testing of gui sequences and their interactions. In: Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on. pp. 54–63. IEEE (2001)